R for Non-Programmers: A Guide for Social Scientists

Daniel Dauber

2021-07-28

Contents

Welcome

Welcome to *R for Non-Programmers: A guide for Social Scientists*. This book is intended to be of help to everyone who wishes to enter the world of R Programming, but not necessarily for the purpose to become a programmer. Instead, this book intends to convey key concepts in data analysis, especially quantitative research.

[Add more]

Acknowledgments

Special thanks are due to my wife, who supported me in so many ways to get this book completed. Also, I would like to thank my son, who patiently watched me sitting at the computer type this book up.

Chapter 1

`Readme.` Before you get started

1.1 A starting point and reference book

`to be added`

1.2 Download the companion R package

`to be added`

- Explain where to download
- Explain how examples are shown in this book (code in the first grey box, console results in the second grey box

1.3 A 'tidyverse' approach with some basic R

`to be added`

Chapter 2

Why learn a programming language as a non-programmer?

'R', it is not just a letter you learn in primary school, but a powerful programming language. While it is used for a lot of quantitative data analysis, it has grown over the years to become a powerful tool that excels (#no-pun-intended) in handling data and performing customised computations with quantitative and qualitative data.

R is now one of my core tools to perform various types of analysis because I can use it in many different ways, for example,

- statistical analysis,
- corpus analysis,
- development of online dashboards to dynamically generate interactive data visualisations,
- connection to social media APIs for data collection,
- Creation of reporting systems to provide individualised feedback to research participants,
- Drafting and writing research articles, etc.

> *Learning R is like learning a foreign language. If you like learning languages, then 'R' is just another one.*

While *R* has become a comprehensive tool for data scientists, it has yet to find its way into the mainstream field of Social Sciences. Why? Well, learning programming languages is not necessarily something that feels comfortable to

everyone. It is not like Microsoft Word, where you can open the software and explore it through trial and error. Learning a programming language is like learning a foreign language: You have to learn vocabulary, grammar and syntax. Similar to learning a new language, programming languages also have steep learning curves and require quite some commitment.

For this reason, most people do not even dare to learn it because it is time-consuming and often not considered a *'core method'* in Social Sciences disciplines. Apart from that, tools like SPSS have very intuitive interfaces, which seem much easier to use (or not?). However, the feeling of having *'mastered' R* (although one might never be able to claim this) can be extremely rewarding.

I guess this introduction was not necessarily helpful in convincing you to learn any programming language. However, despite those initial hurdles, there are a series of advantages to consider. Below I list some good reasons to learn a programming language as they pertain to my own experiences.

2.1 Learning new tools to analyse your data is always essential

Theories change over time, and new insights into certain social phenomena are published every day. Thus, your knowledge might get outdated quite quickly. This is not so much the case for research methods knowledge. Typically, analytical techniques remain over many years. We still use the mean, mode, quartiles, standard deviation, etc., to describe our quantitative data. Still, there are always new computational methods that help us to crunch the numbers even more. *R* is a tool that allows you to venture into new analytical territory because it is open source. Thousands of developers provide cutting-edge research methods free of charge for you to try with your data. You can find them on platforms like GitHub. *R* is like a giant supermarket, where all products are available for free. However, to read the labels on the product packaging and understand what they are, you have to learn the language used in this supermarket.

2.2 Programming languages enhance your conceptual thinking

While I have no empirical evidence for this, I am very certain it is true. While I would argue that my conceptual thinking is quite good, I would not necessarily say that I was born with it. Programming languages are very logical. Any error in your code will make you fail to execute it properly. Sometimes you face challenges in creating the correct code to solve a problem. Through creative abstract thinking (I should copyright this term), you start to approach your problems differently, whether it is a coding problem or a problem in any other

context. For example, I know many students enjoy the process of qualitative coding. However, they often struggle to detach their insights from the actual data and synthesise ideas on an abstract and more generic level. Qualitative researchers might refer to this as challenges in '*second-order deconstruction of meaning'*. This process of abstraction is a skill that needs to be honed, nurtured and practised. From my experience, programming languages are one way to achieve this, but they might not be recognised for this just yet.

2.3 Programming languages allow you to look at your data from a different angle

There are certainly commonly known and well-established techniques regarding how you should analyse your data rigorously. However, it can be quite some fun to try techniques outside your disciplines. This does not only apply to programming languages, of course. Sometimes, learning about a new research method enables you to look at your current tools in very different ways too. One of the biggest challenges for any researcher is to reflect on your work. Learning new and maybe even *'strange'* tools can help with this. Admittedly, sometimes you might find out that some new tools are also a dead-end. Still, you might have learned something valuable through the process of engaging with your data differently. So shake off the rust of your analytical routine and blow some fresh air into your research methods.

2.4 Learning any programming language will help you learn other programming languages.

Once you understand the logic of one language, you will find it relatively easy to understand new programming languages. Of course, if you wanted to, you could become the next '*Neo'* (from 'The Matrix') and change the reality of your research forever. On a more serious note, though, if you know any programming language already, learning R will be easier because you have accrued some basic understanding of these particular types of languages.

Having considered everything of the above, do you feel ready for your next foreign language?

Chapter 3

Setting up R and RStudio

Every journey starts with gathering the right equipment. This intellectual journey is not much different. The first step that every 'R novice has to face is to set everything up to get started. There are essentially two strategies:

- Install *R*and RStudio

or

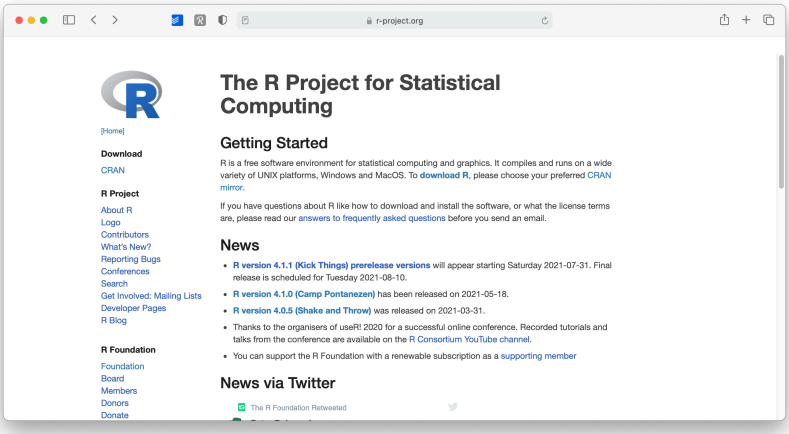
- Run RStudio in a browser via RStudio Cloud

While installing *R* and Studio requires more time and effort, I strongly recommend it, especially if you want to work offline or make good use of your computer's CPU. However, if you are not sure yet whether you enjoy learning *R*, you might wish to look at RStudio Cloud first. Either way, you can follow the examples of this book no matter which choice you make.

3.1 Installing R

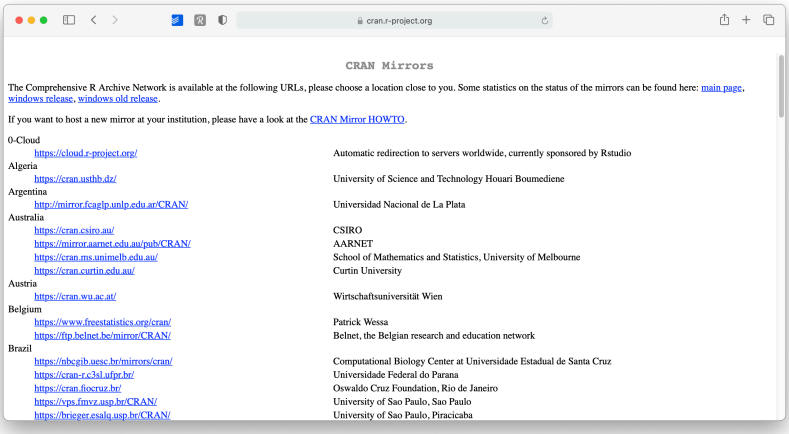
The core module of our programming is R itself, and since it is an open-source project, it is available for free on Windows, Mac and Linux computers. Here is what you need to do to install it properly on your computer of choice:

1. Go to www.r-project.org

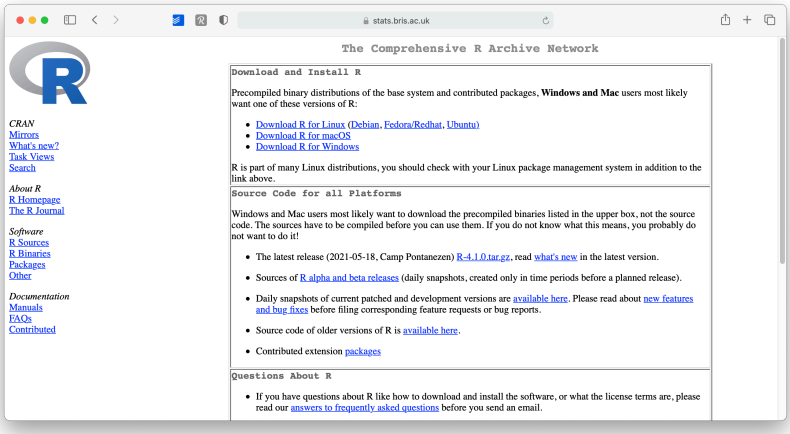


2. Click on `CRAN` where it says download.

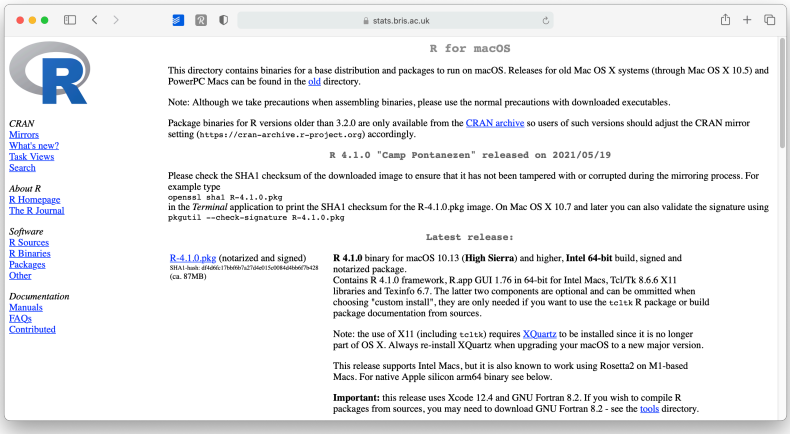
3. Choose a server in your country (all of them work, but downloads will perform quicker).



4. Select the operating system for your computer.



5. Select the version you want to install (I recommend the latest version)



6. Open the downloaded file and follow the installation instructions. (I recommend leaving the suggested settings as they are).

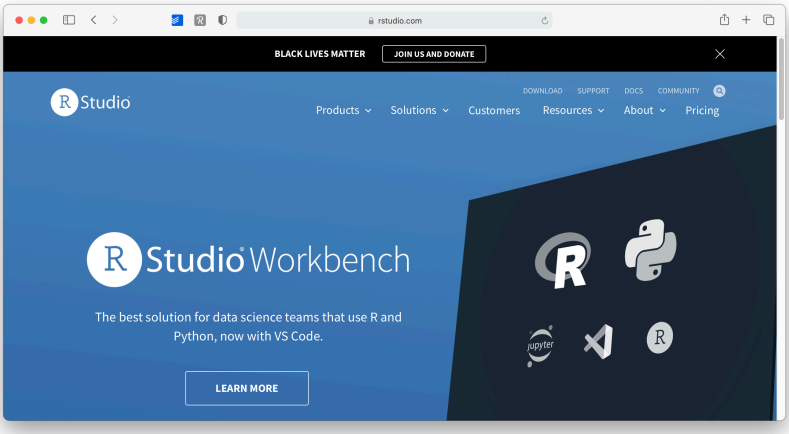
This was relatively easy. You now have *R* installed. Technically you can start using *R* for your research, but there is one more tool I strongly advise installing: RStudio.

3.2 Installing RStudio

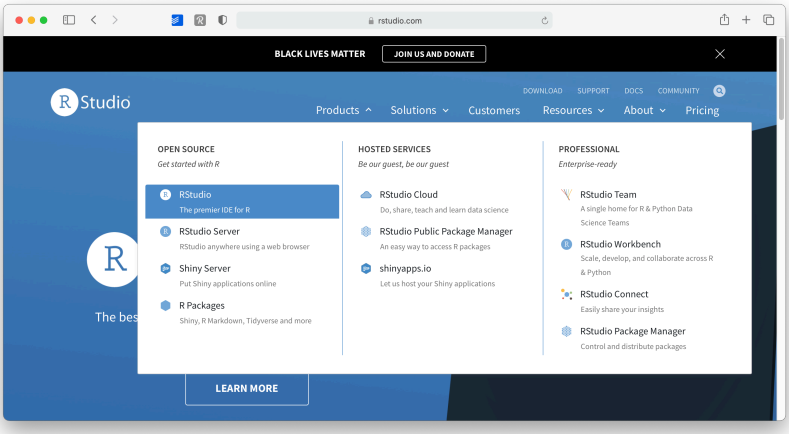
R by itself is just the *'beating heart'* of *R* programming, but it has no particular user interface. If you want buttons to click and actually 'see' what you

are doing, there is no better way than RStudio. RStudio is an *integrated development environment* (IDE) and will be our primary tool to interact with *R*. It is the only software you need to do all the fun parts and, of course, to follow along with the examples of this book. To install RStudio perform the following steps:

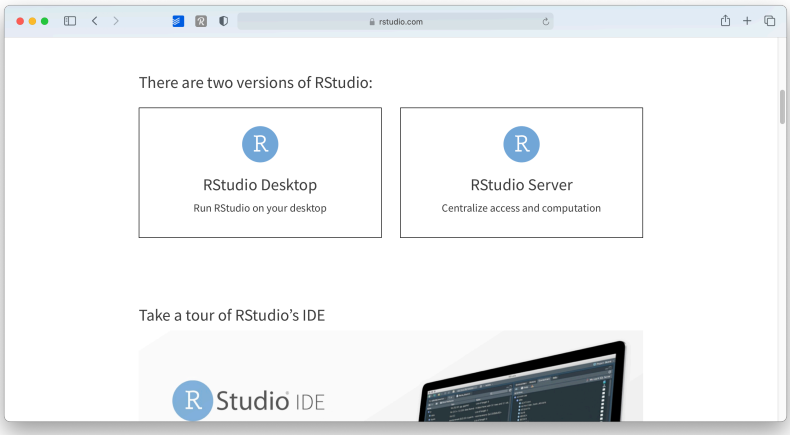
1. Go to www.rstudio.com.



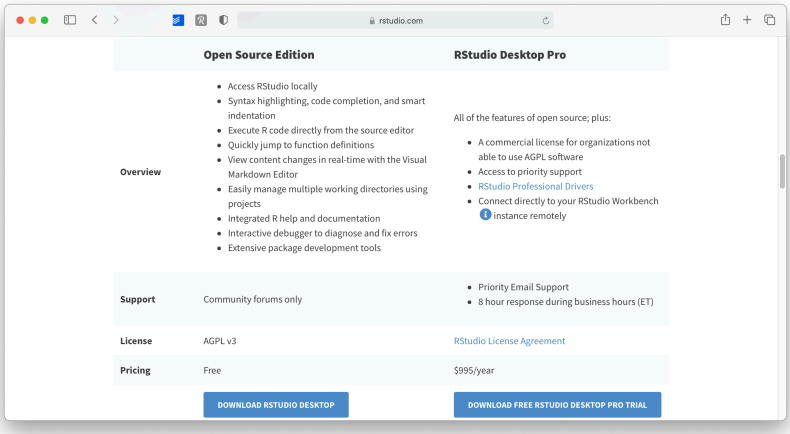
2. Go to `Products > RStudio`



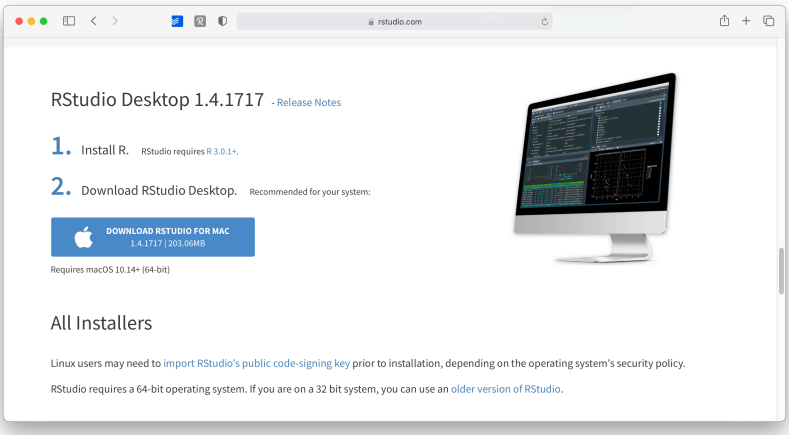
3. On this page, scroll down and select `RStudio Desktop`



4. Select the `'Open Source Edition'` option by clicking on `'Download RStudio Desktop'`



5. As a last step, scroll down where it shows you a download button for your operating system. The website will automatically detect this. You also get a nice reminder to install 'R' first, in case you have not done so yet.



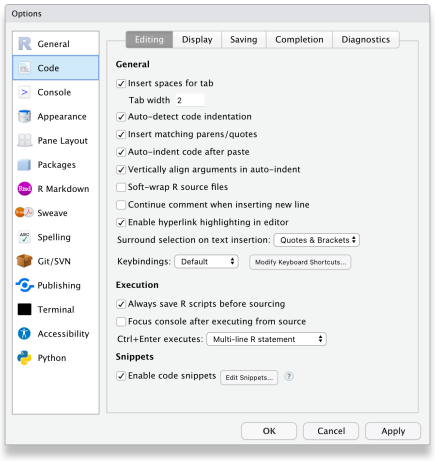
6. Open the downloaded file and follow the installation instructions (again, keep it to the default settings as much as possible)

Congratulations, you are all set up to learn *R*. From now on you only need to start RStudio and not *R*. Of course, if you are the curious, nothing shall stop you to try *R* without RStudio.

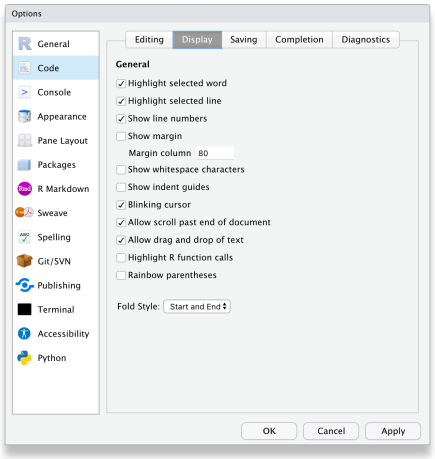
3.3 When you first start RStudio

Before you start programming away, you might want to make some tweaks to your settings right away to have a better experience (in my humble opinion). I recommend at least the following two changes by clicking on `RStudio > Preferences` or press `/Ctrl + ,`.

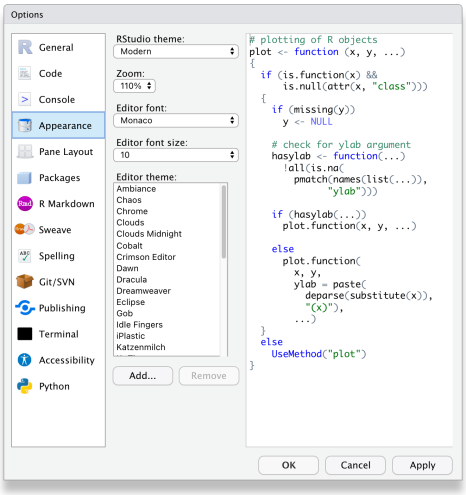
1. In the `Code > Editing` tab, make sure to have at least the first five options ticked, especially the `Auto-indent code after paste`. This setting will save time when trying to format your coding appropriately, making it easier to read. Indentation is the primary way of making your code look more readable and less like a series of characters that appear almost random.



2. In the `Display` tab, you might want to have the first three options selected. In particular, `Highlight selected line` is helpful because, in more complicated code, it is helpful to see where your cursor is.



Of course, if you wish to customise your workspace further, you can do so. The visually most impactful way to alter the default appearance of RStudio is to select Appearance and pick a completely different colour theme. Feel free to browse through various options and see what you prefer. There is no right or wrong here.



3.4 Updating R and RStudio: Living at the pulse of innovation

While not strictly something that helps you become a better programmer, this advice might come in handy to avoid turning into a frustrated programmer. When you update your software, you need to update R and RStudio separately from each other. While both R and RStudio work closely with each other, they still constitute separate pieces of software. Thus, it is essential to keep in mind that updating RStudio will not automatically update R. This can become problematic if specific packages you installed via RStudio (like a fancy learning algorithm) might not be compatible with earlier versions of R. Also, additional R packages developed by other people are separate pieces and are updated too, independently from R and RStudio.

I know what you are thinking: This already sounds complicated and cumbersome. However, rest assured, we take a look at how you can easily update all your packages with RStudio. Thus, all you need to remember is: *R* needs to be updated separately from everything else.

3.5 RStudio Cloud

`to be completed`

Chapter 4

The RStudio Interface

When you open RStudio for the first time, you are presented with four quadrants, each of which fulfils a unique purpose:

- The `Console` window,
- The `Source` window,
- The `Environment / History / Connections / Tutorial` window, and
- The `Files / Plots / Packages / Help / Viewer` window

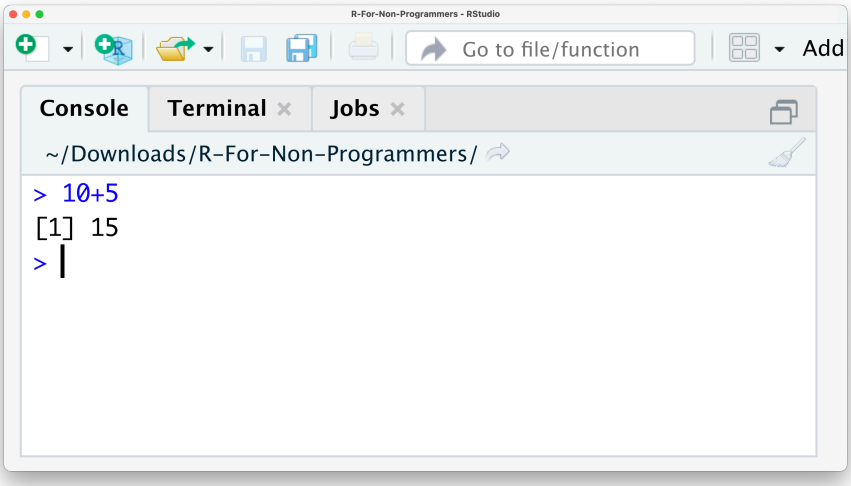
I will briefly explain the purpose of each window/pane and how they are relevant to your work in *R*.

4.1 The Console window

The console is located in the bottom-left, and it is where you often will find the output of your coding and computations. It is also possible to write code directly into the console. Let's try the following example by calculating the sum of `10 + 5`. Click into the console with your mouse, type the calculation into your console and hit `Enter/Return` on your keyboard. The result should be pretty obvious:

```
# We type the below into the console
10+5
## [1] 15
```

Here is a screenshot of how it should look like at your end in RStudio:



You just successfully performed your first successful computation. I know, this is not quite impressive just yet. *R* is undoubtedly more than just a giant calculator.

In the top right of the console, you find a symbol that looks like a broom. This one is quite an important one because it clears your console. Sometimes the console can become very cluttered and difficult to read. If you want to remove whatever you computed, you can click the broom icon and clear the console of all text. I use it so frequently that I strongly recommend learning the keyboard shortcut, which is `Ctrl+L` on PC and Mac.

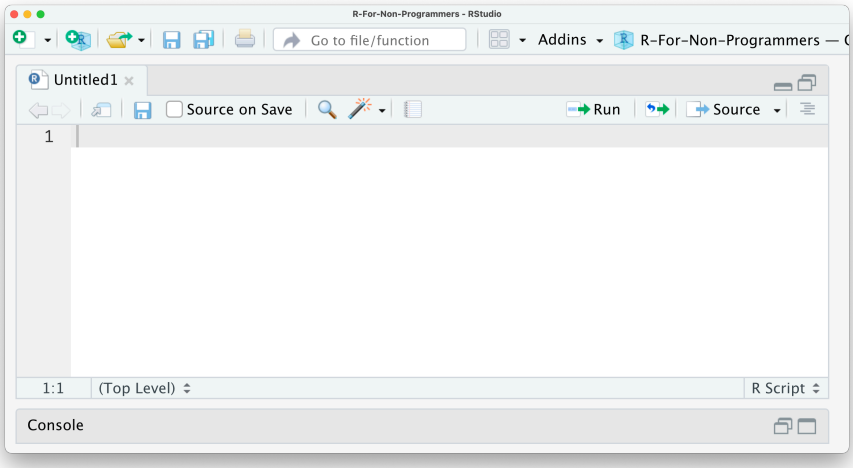
4.2 The Source window

In the top left, you can find the source window. The term 'source' can be understood as any type of file, e.g. data, programming code, notes, etc. The source panel can fulfil many functions, such as:

- Inspect data in an Excel-like format (LINK TO RELEVANT CHAPTER)
- Open programming code, e.g. an R Script (LINK TO RELEVANT CHAPTER)
- Open other text-based file formats, e.g.
  - Plain text (.txt),
  - Markdown (.md),
  - Websites (.html),
  - LaTeX (.tex),

  - BibTex (.bib),

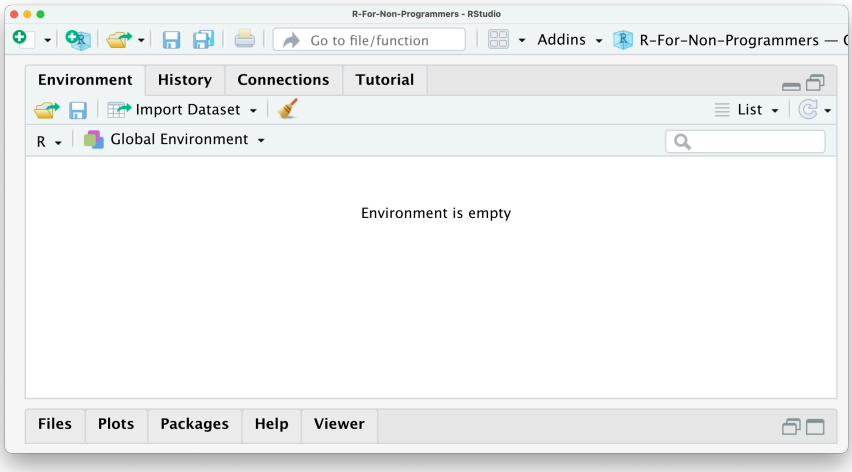
- Edit scripts with code in it,
- Run the analysis you have written.



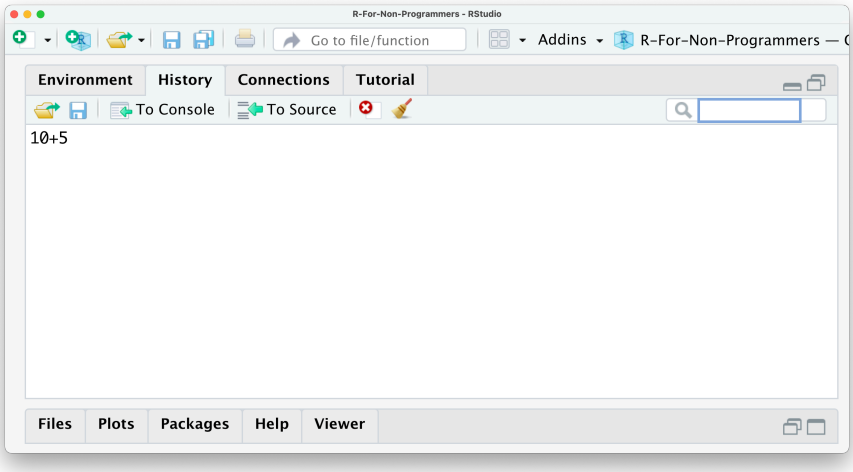
In other words, the source window will show you whatever file you are interested in, as long as RStudio can read it - and no, Microsoft Office Documents are not supported. Another limitation of the source window is that it can only show text-based files. So opening images, etc. would not work.

4.3 The Environment / History / Connections / Tutorial window

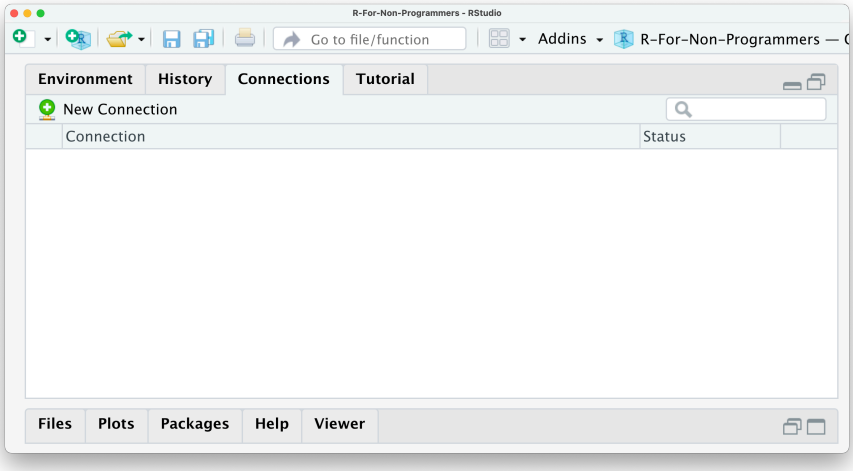
The window in the top right shows multiples panes. The first pane is called *Environment* and shows you objects which are available for computation. One of the first objects you will create is your dataset because, without data, we cannot perform any analysis. Thus, one object might be your data. Another object could be a plot showing the number of male and female participants in your study. To find out how to create objects yourself, you can take a glimpse at (INSERT CHAPTER X). Besides datasets and plots, you will also find other objects here, e.g. lists, vectors and functions you created yourself. Don't worry if none of these words makes sense at this point. We will cover each of them in the upcoming chapters. For now, remember this is a place where you can find different objects you created.



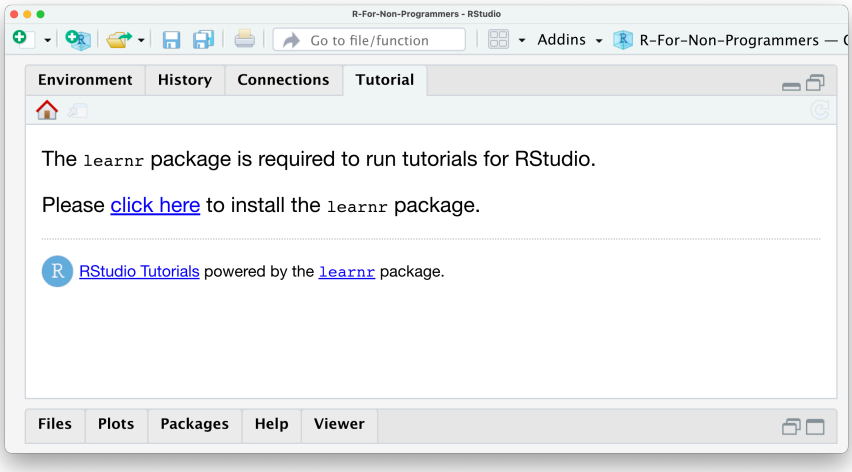
The *History* pane is very easy to understand. Whatever computation you run in the Console will be stored. So you can go back and see what you coded and rerun that code. Remember the example from above where we computed the sum of `10+5`? This computation is stored in the history of RStudio, and you can rerun it by clicking on `10+5` in the history pane and then click on `To Console`. This will insert `10+5` back into the Console, and we can hit `Return` to retrieve the result. You also have the option to copy the code into an existing or new R Script by clicking on `To Source`. By doing this, you can save this computation on your computer and reuse it later. Finally, if you would like to store your history, you can do so by clicking on the `floppy disk symbol`. There are two more buttons in this pane, one allows you to delete individual entries in the history, and the last one, a `broom`, clears the entire history (irrevocably).



The pane *Connections* allows you to tab into external databases directly. This can come in handy when you work collaboratively on the same data or want to work with extensive datasets without having to download them. However, for an introduction to R, we will not use this feature of RStudio for now.

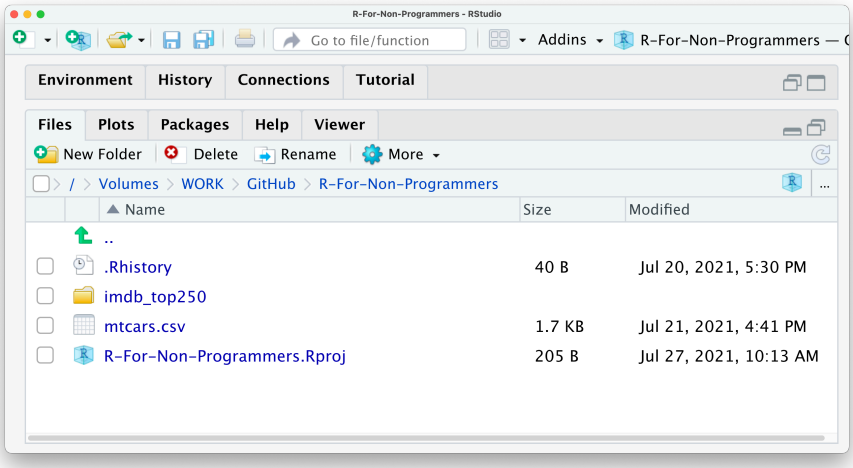


The last pane is called *Tutorial*. Here you can find additional materials to learn *R* and RStudio. If you search for more great content to learn R, this serves as a great starting point.



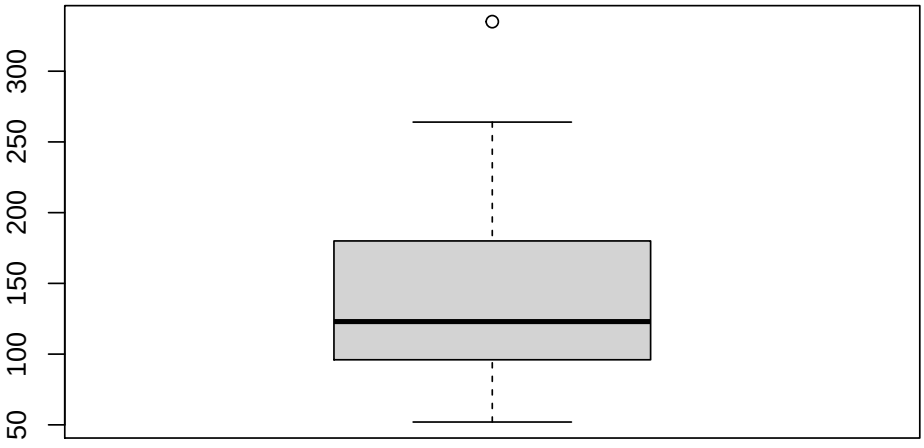
4.4 The Files / Plots / Packages / Help / Viewer window

The last window consists of five essential panes. The first one is the *Files* pane. As the name indicates, it lists all the files and folders in your root directory. A root directory is the default directory where RStudio saves your files, for example, your analysis. However, you can easily change this directory to something else (see also CHAPTER X) or use R Project files (see CHAPTER X) to carry out your research. Thus, the *Files* pane is an easy way to load data into RStudio and create folders to keep your research project well organised.



Since the Console cannot reproduce data visualisations, RStudio offers a way to do this very easily. It is through the Plots pane. This pane is exclusively designed to show you any plots you have created using R. Here is a simple example that you can try. Type into your console `boxplot(mtcars$hp)`.

```
# Here we create a nice boxplot using a dataset called 'mtcars'
boxplot(mtcars$hp)
```

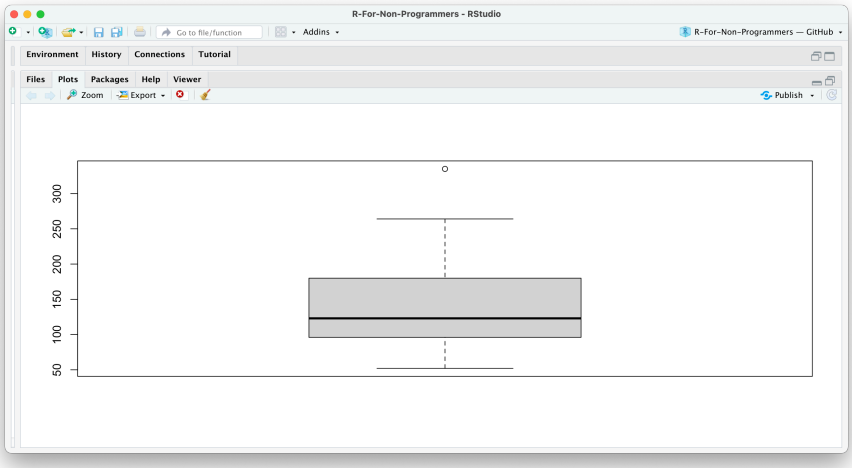


Although this is a short piece of coding, it performs quite a lot of steps:

- it uses a function called `boxplot()` to draw a boxplot of
- a variable called `hp` (for horsepower), which is located in
- a dataset named `mtcars`,

- and it renders the graph in your *Plots* pane

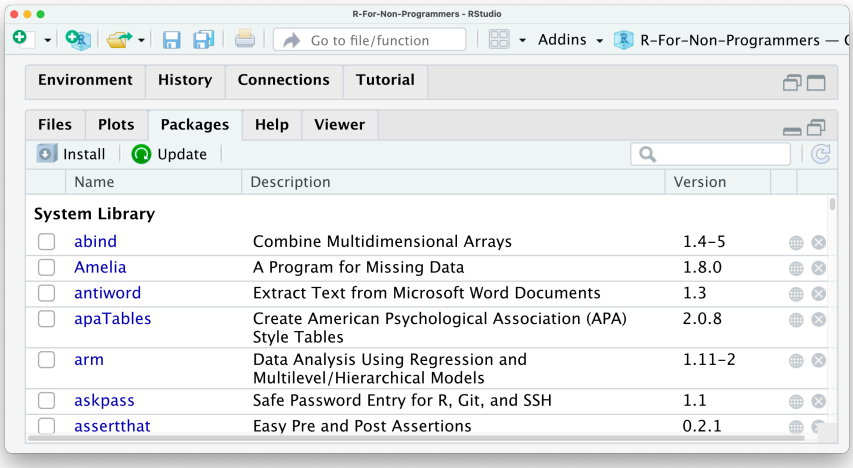
This is how the plot should look like in your RStudio *Plots* pane.



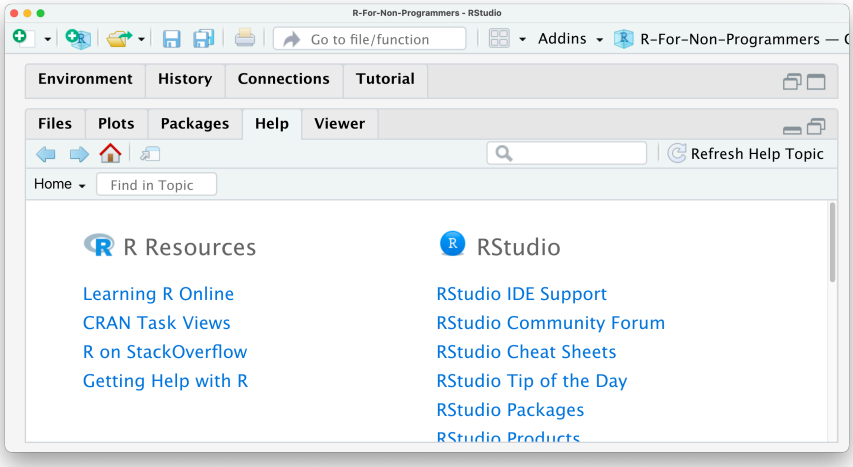
If you wish to delete the plot, you can click on the `red circle with a white x` symbol. This will delete the currently visible plot. If you wish to remove all plots from this pane, you can use the `broom`. There is also an option to export your plot and move back and forth between different plots.

Do not worry about the coding at this point. It will all make sense in the following chapters.

The next pane is called *Packages*. Packages are additional tools you can import and use when performing your analysis. A frequent analogy people use to explain packages is your phone and the apps you install. Each package you download is equivalent to an app on your phone. It can enhance different aspects of working in *R*, such as creating animated plots, using unique machine learning algorithms, or simply making your life easier by doing multiple computations with just one single line of code. You will learn more about *R packages* in Chapter 5.4.



If you are in dire need of help, RStudio provides you with a *Help* pane. You can search for specific topics, for example how certain computations work. The *Help* pane also has documentation on different datasets that are included in *R*, RStudio or *R packages* you have installed.

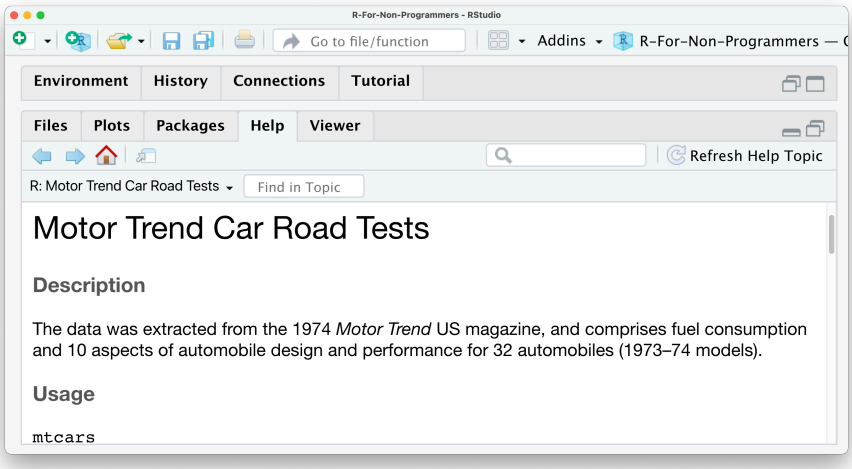


So, for example, if you want to know what the `mtcars` dataset is, you can either use the search window in the *Help* pane or, much easier, use a `?` in the console to search for it:

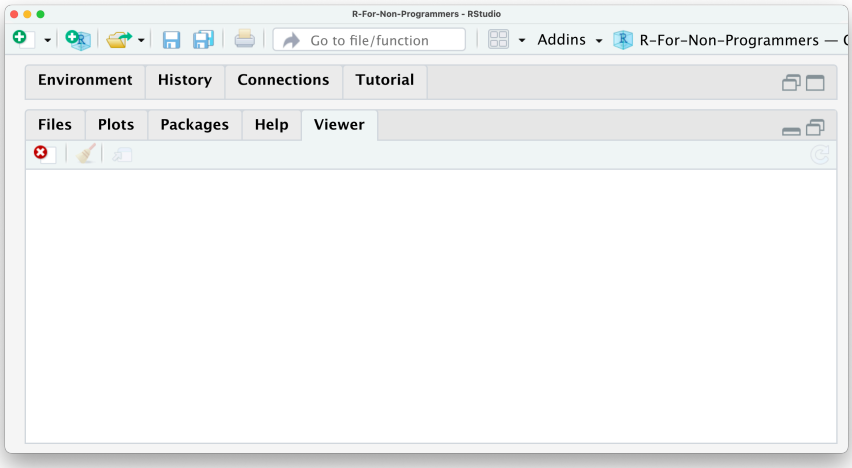
```
# Type a '?' and immediately add the name to bring up helpful information.

?mtcars
```

This will open the *Help* pane and give you more information about this dataset:



Lastly, we have the *Viewer* pane. Not every data visualisation we create in R is a static image. You can create dynamic data visualisations or even websites with R. This type of content is displayed in the Viewer pane rather than in the Plots pane. Often these visualisations are based on HTML and other web-based programming languages. As such, it is easy to open them in your browser as well. However, in this book, we mainly focus on two-dimensional static plots, which are the ones you likely need most of the time, either for your assignments, thesis, or publication.



4.5 Customise your user interface

As a last remark in this chapter, I would like to make you aware that you can modify each window. There are three basic adjustments you can make:

- Hide panes by clicking on the window symbol in the top right corner of each window,
- Resize panes by dragging the border of a window horizontally or vertically, or
- Add and remove panes by going to `RStudio > Preferences > Pane Layout`, or use the keyboard shortcut `+ ,` if you are on a Mac. There is, unfortunately no default shortcut for PC users.

Chapter 5

R Basics: The very fundamentals

After a likely tedious installation of R and RStudio, as well as a somewhat detailed introduction to the RStudio interface, you are finally ready to 'do' things. By 'doing', I mean coding. The term 'coding' in itself can instil fear in some of you, but you only need one skill to do it: Writing. As mentioned earlier, learning coding or programming means learning a new language. However, once you have the basic grammar down, you already can communicate quite a bit. In this section, we will explore the fundamentals of R. These build the foundation for everything that follows. After that, we dive right into some analysis.

5.1 Basic computations in R

The most basic computation you can do in R is arithmetic operations. In other words, addition, subtraction, multiplication, division, exponentiation and extraction of roots. In other words, R can be used like your pocket calculator, or more likely the one you have on your phone. For example, in Chapter 4.1 we already performed an addition. Thus, it might not come as a surprise how their equivalents work in R. Let's take a look at the following examples:

```
# Addition
10+5
## [1] 15

# Subtraction
10-5
## [1] 5
```

```
# Multiplication
10*5
## [1] 50

# Division
10/5
## [1] 2

# Exponentiation
10^2
## [1] 100

# Square root
sqrt(10)
## [1] 3.162278
```

They all look fairly straightforward except for the extraction of roots. As you probably know, extracting the root would typically mean we use the symbol $\sqrt{}$ on your calculator. To compute the square root in R, we have to use a function instead to perform the computation. So we first put the name of the function `sqrt` and then the value `10` within parenthesis `()`. This results in the following code: `sqrt(10)`. If we were to write this down in our report, we would write $\sqrt[2]{10}$.

Functions are an essential part of R and programming in general. You will learn more about them in this chapter.Besides arithmetic operations, there are also logical queries you can perform. Logical queries always return either the value TRUE or FALSE. Here are some examples which make this clearer:

```
#1 Is it TRUE or FALSE?
1 == 1
## [1] TRUE

#2 Is 45 bigger than 55?
45 > 55
## [1] FALSE

#3 Is 1982 bigger or equal to 1982?
1982 >= 1982
## [1] TRUE

#4 Are these two words NOT the same?
"Friends" != "friends"
## [1] TRUE
```

```
#5 Are these sentences the same?
"I love statistics" == "I love statistícs"
## [1] FALSE
```

Reflecting on these examples, you might notice three important things:

1. I used `==` instead of `=`,
2. I can compare non-numerical values, i.e. text, which is also known as `character values`, with each other,
3. The devil is in the details considering #5.

One of the most common mistakes of R novices is the confusion around the `==` and `=` notation. While `==` represents `equal to`, `=` is used to assign a value to an object (for more details on assignments see Chapter 4.4). However, in practice, most R programmers tend to avoid `=` since it can easily lead to confusion with `==`. As such, you can strike this one out of your R vocabulary for now.

There are many different logical operations you can perform. Table 5.1 lists the most frequently used logical operators for your reference. These will become important once we select only certain parts of our data for analysis, e.g. only `female` participants.

Table 5.1: Logical Operators in R

Operator	Description
==	is equal to
>=	is bigger or equal to
<=	is smaller of equal to
!=	is not equal to
a | b	a or b
a & b	a and b
!a	is not a

5.2 Assigning values to objects: '<-'

Another common task you will perform is assigning values to an object. An object can be many different things:

- a dataset,
- the results of a computation,
- a plot,

- a series of numbers,
- a list of names,
- a function,
- etc.

In short, an object is an umbrella term for many different things which form part of your data analysis. For example, objects are handy when storing results that you want to process further in later analytical steps. Let's have a look at an example.

```
# I have a friend called "Fiona"
friends <- "Fiona"
```

In this example, I created an object called `friends` and added `"Fiona"` to it. Remember, because `"Fiona"` represents a `string`, we need `""`. So, if you wanted to read this line of code, you would say, '`friends` gets the value `"Fiona"`'. Alternatively, you could also say '`"Fiona"` is assigned to `friends`'.

If you look into your environment pane, you will find the object we just created. You can see it carries the value `"Fiona"`. We can also print values of an object in the console by simply typing the name of the object `friends` and hit `Return` .

```
# Who are my friends?
friends
```

```
## [1] "Fiona"
```

Sadly, it seems I only have one friend. Luckily we can add some more, not the least to make me feel less lonely. To create objects with multiple values, we can use the function `c()`, which stands for 'concatenate'. The Cambridge Dictionary (2021) define this word as follows:

> '***concatenate***',
> to put things together as a connected series

Let's concatenate some more friends into our `friends` object.

```
# Adding some more friends to my life
friends <- c("Fiona", "Ida", "Lukas", "Georg", "Daniel", "Pavel", "Tigger")

# Here are all my friends
friends
## [1] "Fiona"  "Ida"    "Lukas"  "Georg"  "Daniel" "Pavel"  "Tigger"
```

To concatenate values into a single object, we need to use a comma `,` to separate each value. Otherwise, R will report an error back.

```
friends <- c("Fiona" "Ida")
## Error: <text>:1:22: unexpected string constant
## 1: friends <- c("Fiona" "Ida"
##                          ^
```

R's error messages tend to be very useful and give meaningful clues to what went wrong. In this case, we can see that something 'unexpected' happen, and it shows where our mistake is.

You can also concatenate numbers, and if you add `()` around it, you can automatically print the content of the object to the console. Thus, `(milestones_of_my_life <- c(1982, 2006, 2011, 2018, 2020))` is the same as `milestones_of_my_life <- c(1982, 2006, 2011, 2018, 2020)` followed by `milestones_of_my_life`. The following examples illustrate this.

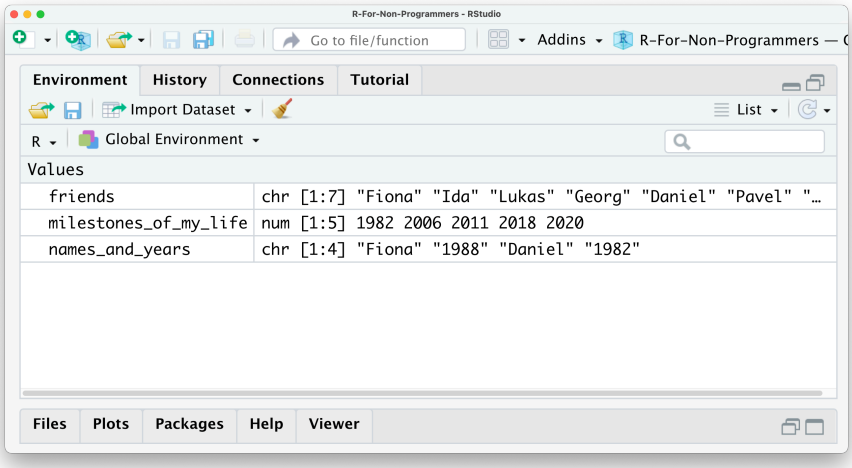
```
# Important years in my life
milestones_of_my_life <- c(1982, 2006, 2011, 2018, 2020)
milestones_of_my_life
## [1] 1982 2006 2011 2018 2020

# The same as above, but we don't need the second line of code
(milestones_of_my_life <- c(1982, 2006, 2011, 2018, 2020))
## [1] 1982 2006 2011 2018 2020
```

Finally, we can also concatenate numbers and character values into one object:

```
(names_and_years <- c("Fiona", 1988, "Daniel", 1982))
## [1] "Fiona"  "1988"   "Daniel" "1982"
```

This last example is not necessarily something I would recommend to do, because it likely leads to undesirable outcomes. If you look into your environment pane you currently have three objects: `friends`, `milestones_of_my_life`, and `names_and_years`.



The `friends` object shows that all the values inside the object are classified as `chr`, which denominates `character`. In this case, this is correct because it only includes the names of my friends. On the other hand, the object `milestones_of_my_life` only includes `numeric` values, and therefore it says `num` in the environment pane. However, for the object `names_and_years` we know we want to have `numeric` and `character` values included. Still, R recognises them as `character` values only because values inside objects are meant to be of the same type.

Consequently, mixing different types of data (as explained in Chapter @ref()) into one object is likely a bad idea. This is especially true if you want to use the numeric values for computation. In short: ensure your objects are all of the same data type.

There is an exception to this rule. 'Of course', you might say. There is one object that can have values of different types: list. As the name indicates, a list object holds several items. These items are usually other objects. In the spirit of 'Inception', you can have lists inside lists, which contain more objects.

Let's create a list called `x_files` using the `list` function and place all our objects inside.

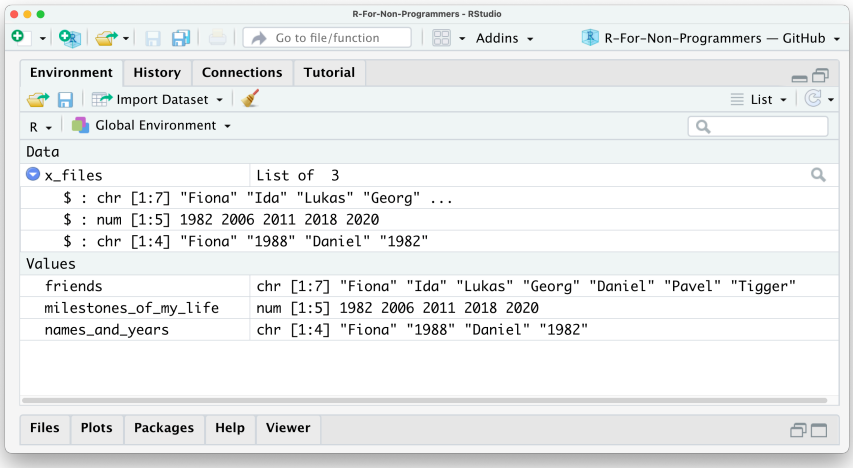
```
# This creates our list of objects
x_files <- list(friends,
                milestones_of_my_life,
                names_and_years)

# Let's have a look what is hidden inside the x_files
x_files
## [[1]]
```

```
## [1] "Fiona"  "Ida"    "Lukas"  "Georg"  "Daniel" "Pavel"  "Tigger"
##
## [[2]]
## [1] 1982 2006 2011 2018 2020
##
## [[3]]
## [1] "Fiona"  "1988"   "Daniel" "1982"
```

You will notice in this example that I do not use `""` for each value in the list. This is because `friends` is not a character I put into the list, but an object. When we refer to objects we do not need the quotation marks.

We will encounter lists quite frequently when we perform our analysis. Some functions return the results in the format of lists. This can be very helpful, because otherwise our environment pane will be littered with objects, but we would not necessarily know how they relate to each other or worse, to which analysis they belong. Looking at the list item in the environment page, you can see that the object `x_files` is classified as a `List of 3` and if you click on the blue icon, you can inspect the different objects inside.



In Chapter 5.1 I mentioned that we should avoid using the `=` operator and explained that it is used to assign values to objects. You can, if you want, use `=` instead of `<-`. They fulfil the same purpose. However, as mentioned before, it is not wise to do so. Here is an example that shows that, in principle, it is possible.

```
# DO
(avengers1 <- c("Iron Man", "Captain America", "Black Widow", "Vision"))
## [1] "Iron Man"        "Captain America" "Black Widow"     "Vision"
```

```
# DON'T
(avengers2 = c("Iron Man", "Captain America", "Black Widow", "Vision"))
## [1] "Iron Man"        "Captain America" "Black Widow"     "Vision"
```

On a final note, naming your objects is limited. Not every name can be chosen. First, every name needs to start with a letter. You also can only use letters, numbers `_` and `.` as valid components of the names of your objects (see also Wickham and Grolemund, 2016, Chapter 4.2.).

5.3 Functions

I used the term 'function' multiple times already, but I never really fully explained what they are and why we need them. In very simple terms, functions are objects. They contain lines of code that someone has written for us (ro we have written ourselves). One could say they are code snippets ready to use. Someone else might see them as shortcuts for our programming. Functions not only increase the speed with which we perform our analysis and write our computations, but also make our code more readable. Consider computing the `mean` of values stored in the object `pocket_money`.

```
# First we create an object that stores our desired values
pocket_money <- c(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89)

#1 Manually compute the mean:
sum <- 0+1+1+2+3+5+8+13+21+34+55+89
sum/12 # There are 12 items in the object
## [1] 19.33333

#2 Use a function to compute the mean:
mean(pocket_money)
## [1] 19.33333

#3 Let's make sure it is actually the same
sum/12 == mean(pocket_money)
## [1] TRUE
```

If we manually compute the mean, we first compute the sum of all values in the object `pocket_money`[1], and then we divide it by the number of values in the object, which is `12`. This is the classic way of computing the mean as we know it. However, my simply using the function `mean()`, we not only write

[1] If you find the order of numbers suspicious, it is because it represents the famous Fibonacci sequence.

considerably less code, it is much easer to understand as well, because the word `mean` does exactly what we would expect. Which one do you find easier?

All functions in R share the same structure. They have a `name` follow by `()` and within these parentheses we put `arguments`, which have certain `values`. A function would look something like this:

```
name_of_function(argument_1 = value_1,
                 argument_2 = value_2,
                 argument_3 = value_3)
```

How many arguments there are and what kind of values you can provide is very much dependent on the function you use. Thus, not every function takes every value. It is fair to make the analogy that function names are similar to vocabulary of a foreign language, while the arguments represent a specific syntax that goes with it. In the case of `mean()`, the function takes an object which holds a sequence of `numeric` values. It would make very little sense to compute the mean of our `friends` object, because it only contains names. R would return an error message:

```
mean(friends)
## Warning in mean.default(friends): argument is not numeric or logical: returning
## NA
## [1] NA
```

`NA` refers to a value that is *'not available'*. In this case, R tries to compute the mean but the result is not available. However, it can run the function `mean()`. Therefore, this value is `NA`. In your dataset, you might find cells that are `NA`, which means there is data missing. You will learn more about how to deal with this in Chapter @ref().

Sometimes you will also get a message from R that something is `NaN`. This stands for *'not a number'* and is returned when something is not possible to compute, for example:

```
# Example 1
0/0
## [1] NaN

# Example 2
sqrt(-9)
## Warning in sqrt(-9): NaNs produced
## [1] NaN
```

5.4 R packages

R has many built-in functions that we can use right away. However, some of the most interesting ones are actually developed by other programmers, data scientists and enthusiasts. To add more functions to your repertoire, you can install R packages. R packages are a collection of functions that you can download and use for your own analysis. Throughout this book you will learn about and use many different R packages to accomplish different tasks.

To give you another analogy,

- R is like a global supermarket,
- RStudio is like my shopping cart,
- and R packages are the products I can pick from the shelves.

Luckily, R packages are free to use, so I do not have to bring my credit card. For me, these additional functions, developed by some of the greatest scientists, is what keeps me addicted to performing my research in R.

However, how do you find those R packages? They are right at your fingertips. You have mainly two options:

1. Use the Packages pane (see Chapter 4.4)
2. Or call a function

5.4.1 Installing packages via RStudio's Package pane

`to be added`

5.4.2 Installing packages using a function

The simplest and fastest way to install a package is calling the function `install.packages()`. You can either use it to install a single package or install a series of packages all at once using our trusty `c()` function.

```
# Install a single package
install.packages("tidyverse")

# Installe multiple packages at once
install.packages(c("tidyverse", "naniar", "psych"))
```

5.5 Exercises

1. What is the result of $\sqrt[2]{25 - 16} + 2 * 8 - 6$?
2. What does the console return if you execute the following code `"Five" == 5`?
3. Create a list called `books` and include the folKlowing book titles in it:
    - "Harry Potter and the Deathly Hallows",
    - "The Alchemist",
    - "The Davinci Code",
    - "R For Dummies"

Check you answers: 6.1

Chapter 6

Exercises: Solutions

6.1 Solutions for ??

```
#1
sqrt(25-16)+2*8-6
## [1] 13

#2
"Five" == 5
## [1] FALSE

#3
books <- list("Harry Potter and the Deathly Hallows",
              "The Alchemist",
              "The Davinci Code",
              "R For Dummies")

books
## [[1]]
## [1] "Harry Potter and the Deathly Hallows"
##
## [[2]]
## [1] "The Alchemist"
##
## [[3]]
## [1] "The Davinci Code"
##
## [[4]]
## [1] "R For Dummies"
```

Bibliography

Cambridge Dictionary (2021). Concatenate. Accessed: 2021-07-28.

Wickham, H. and Grolemund, G. (2016). *R for data science: import, tidy, transform, visualize, and model data.* " O'Reilly Media, Inc.".